Índice

1. Introducción: ¿Qué es MapStruct y por qué usarlo?

Concepto

MapStruct es una librería de mapeo Java Bean a Java Bean que genera código de conversión en tiempo de compilación. Su función principal es transformar objetos entre distintas capas, por ejemplo:

- Entidades JPA ↔ DTOs (Data Transfer Objects)
- Modelos de dominio ↔ Objetos de vista
- Tipos complejos ↔ Representaciones simplificadas

Ventajas

- Rendimiento: el código se genera en compilación, sin reflexión.
- Seguridad de tipo: los errores se detectan al compilar.
- Integración nativa con Spring (**componentModel = "spring"**).
- Legibilidad y mantenimiento: reduce código repetitivo de mapeo manual.

2. Requisitos técnicos

Tecnología	Versión recomendada
Java	17 o superior
Spring Boot	3.x
MapStruct	1.6.15.Final
Lombok	1.18.30
Maven	3.8+
Base de datos	H2 (para pruebas)

3. Configuración del proyecto

Estructura recomendada del proyecto

```
src/main/java/com/example/mapstructdemo/
  ├── controller/
  ├── dto/
  ├── entity/
  ├── mapper/
  ├── repository/
  ├── service/
  └── MapstructDemoApplication.java
```

3.1 Dependencias Maven

```
dependencies>

    <!-- Spring Boot Core -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.30</version>
        <scope>provided</scope>
    </dependency>
```

```
    <!-- MapStruct -->
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>1.6.15.Final</version>
    </dependency>

    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-processor</artifactId>
        <version>1.6.15.Final</version>
        <scope>provided</scope>
    </dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.1</version>
            <configuration>
                <source>17</source>
                <target>17</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>1.6.15.Final</version>
                    </path>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>1.18.30</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

3.2 Configuración H2 (application.properties)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
spring.h2.console.enabled=true
spring.jpa.show-sql=true
```

4. Ejemplo base: Entidades y DTOs

Creamos un modelo sencillo con relaciones:
Un **Customer** puede tener varios **Order**, y cada **Order** se asocia a un **Product**.

4.1 Entidades

Customer.java

```
package com.example.mapstructdemo.entity;

import jakarta.persistence.*;
import lombok.Data;
import java.util.List;

@Entity
@Data
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    @OneToMany(mappedBy = "customer")
    private List<Order> orders;
}
```

Product.java

```java
package com.example.mapstructdemo.entity;

import jakarta.persistence.*;
import lombok.Data;

@Entity
@Data
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private Double price;
}
```

Order.java

```java
package com.example.mapstructdemo.entity;

import jakarta.persistence.*;
import lombok.Data;

@Entity
@Data
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Double total;

    @ManyToOne
    private Customer customer;

    @ManyToOne
    private Product product;
}
```

4.2 DTOs

OrderSummaryDTO.java

```java
package com.example.mapstructdemo.dto;

import lombok.Data;

@Data
public class OrderSummaryDTO {
    private Long orderId;
    private String productName;
    private Double total;
}
```

CustomerDTO.java

```java
package com.example.mapstructdemo.dto;
import lombok.Data;
import java.util.List;

@Data
public class CustomerDTO {
    private Long id;
    private String name;
    private String email;
    private List<OrderSummaryDTO> orders;
}
```

ProductDTO.java

```java
package com.example.mapstructdemo.dto;
import lombok.Data;

@Data
public class ProductDTO {
    private Long id;
    private String name;
    private Double price;
}
```

OrderDetailDTO.java

```
package com.example.mapstructdemo.dto;
import lombok.Data;

@Data
public class OrderDetailDTO {
    private Long orderId;
    private String customerName;
    private String productName;
    private Double total;
}
```

5. Mappers: desde básicos hasta avanzados

5.1 ProductMapper.java

```
package com.example.mapstructdemo.mapper;

import org.mapstruct.Mapper;
import com.example.mapstructdemo.dto.ProductDTO;
import com.example.mapstructdemo.entity.Product;

@Mapper(componentModel = "spring")
public interface ProductMapper {
    ProductDTO toDTO(Product product);
    Product toEntity(ProductDTO dto);
}
```

5.2 OrderMapper.java

```
package com.example.mapstructdemo.mapper;

import org.mapstruct.*;
import com.example.mapstructdemo.dto.*;
import com.example.mapstructdemo.entity.*;

@Mapper(componentModel = "spring", uses = ProductMapper.class)
public interface OrderMapper {
```

```
    @Mapping(source = "id", target = "orderId")
    @Mapping(source = "product.name", target = "productName")
    @Mapping(source = "total", target = "total")
    OrderSummaryDTO toSummaryDTO(Order order);

    @Mapping(source = "id", target = "orderId")
    @Mapping(source = "customer.name", target = "customerName")
    @Mapping(source = "product.name", target = "productName")
    OrderDetailDTO toDetailDTO(Order order);
}
```

5.3 CustomerMapper.java

```
package com.example.mapstructdemo.mapper;

import org.mapstruct.*;
import com.example.mapstructdemo.dto.*;
import com.example.mapstructdemo.entity.*;
import java.util.List;

@Mapper(componentModel = "spring", uses = { OrderMapper.class })
public interface CustomerMapper {

    @Mapping(source = "orders", target = "orders")
    CustomerDTO toDTO(Customer customer);

    List<CustomerDTO> toDTOList(List<Customer> customers);
}
```

6. Servicios y controladores

6.1 Repositorios

```
public interface CustomerRepository extends JpaRepository<Customer,
Long> {}
public interface OrderRepository extends JpaRepository<Order, Long> {}
public interface ProductRepository extends JpaRepository<Product, Long>
{}
```

6.2 Servicio

```java
@Service
@RequiredArgsConstructor
public class CustomerService {

    private final CustomerRepository customerRepository;
    private final CustomerMapper customerMapper;

    public List<CustomerDTO> getAll() {
        return customerMapper.toDTOList(customerRepository.findAll());
    }

    public CustomerDTO getById(Long id) {
        return customerRepository.findById(id)
                .map(customerMapper::toDTO)
                .orElseThrow(() -> new RuntimeException("Customer not found"));
```

6.3 Controlador REST

```java
@RestController
@RequestMapping("/api/customers")
@RequiredArgsConstructor
public class CustomerController {

    private final CustomerService customerService;

    @GetMapping
    public List<CustomerDTO> getAll() {
        return customerService.getAll();
    }

    @GetMapping("/{id}")
    public CustomerDTO getById(@PathVariable Long id) {
        return customerService.getById(id);
    }
}
```

7. Mapeos avanzados y técnicas profesionales

Técnica	Descripción	Ejemplo
@Named + qualifiedByName	Conversiones personalizadas	Fecha **String → LocalDate**
@AfterMapping	Lógica post-mapeo	Calcular **priceWithTax**
@BeforeMapping	Normalizar datos antes de mapear	Limpiar espacios
uses = {Mapper.class}	Reutilizar otros mappers	Mappers modulares
@MappingTarget	Modificar objeto destino	Enriquecer DTO existente
defaultValue / ignore	Controlar campos nulos o no usados	**@Mapping(target="id", ignore=true)**

7.1 @Named + qualifiedByName

Objetivo:
Realizar conversiones personalizadas de tipos de datos no compatibles directamente, como **String ↔ LocalDate, Integer ↔ Boolean**, o conversiones con formatos especiales.

Cómo funciona:

- **@Named** se usa para nombrar un método auxiliar de conversión.
- **qualifiedByName** se usa en un **@Mapping** para referirse a ese método concreto.

Ejemplo: convertir **String ↔ LocalDate**

DTO y Entidad:

```
@Data
public class EmployeeDTO {
    private Long id;
    private String name;
    private String birthDate; // "2025-11-19"
}
```

```java
@Entity
@Data
public class Employee {
    @Id
    private Long id;
    private String name;
    private LocalDate birthDate;
}
```

Mapper:

```java
@Mapper(componentModel = "spring")
public interface EmployeeMapper {

    @Mapping(source = "birthDate", target = "birthDate", qualifiedByName = "stringToLocalDate")
    Employee toEntity(EmployeeDTO dto);

    @Mapping(source = "birthDate", target = "birthDate", qualifiedByName = "localDateToString")
    EmployeeDTO toDTO(Employee entity);

    @Named("stringToLocalDate")
    default LocalDate stringToLocalDate(String date) {
        return date != null ? LocalDate.parse(date) : null;
    }

    @Named("localDateToString")
    default String localDateToString(LocalDate date) {
        return date != null ? date.toString() : null;
    }
}
```

Resultado:

- MapStruct llama automáticamente al método anotado con **@Named** cuando usa **qualifiedByName**.
- Permite reutilizar conversiones complejas en múltiples mappers.

7.2 @AfterMapping

Objetivo:
Ejecutar lógica posterior al mapeo automático, cuando necesitas realizar cálculos o asignar campos derivados después de que MapStruct haya hecho su trabajo principal.

Cómo funciona:

- Se declara un método con **@AfterMapping**.
- Recibe el objeto de origen y el de destino mediante **@MappingTarget**.
- Puedes modificar el resultado final antes de devolverlo.

Ejemplo: calcular **priceWithTax**

```java
@Data
public class Product {
    private String name;
    private Double price;
}

@Data
public class ProductDTO {
    private String name;
    private Double price;
    private Double priceWithTax;
}

@Mapper(componentModel = "spring")
public interface ProductMapper {

    ProductDTO toDTO(Product product);

    @AfterMapping
    default void addTax(Product product, @MappingTarget ProductDTO dto) {
        dto.setPriceWithTax(product.getPrice() * 1.21); // 21% IVA
    }
}
```

Resultado:

- **priceWithTax** se calcula automáticamente tras mapear.
- Útil para añadir campos derivados o calculados sin alterar la entidad base.

7.3 @BeforeMapping

Objetivo:
Ejecutar lógica antes del mapeo, generalmente para normalizar o limpiar datos antes de que MapStruct los use.

Cómo funciona:

- Igual que **@AfterMapping**, pero se ejecuta antes del proceso de mapeo.
- Ideal para limpiar cadenas, validar formatos, etc.

Ejemplo: normalizar texto

```
@Mapper(componentModel = "spring")
public interface UserMapper {

    UserDTO toDTO(User user);

    @BeforeMapping
    default void trimFields(User user) {
        if (user.getName() != null)
            user.setName(user.getName().trim());
        if (user.getEmail() != null)
            user.setEmail(user.getEmail().toLowerCase());
    }
}
```

Resultado:

- Se asegura que los datos están limpios y normalizados antes de convertirlos en DTO.
- Útil cuando recibes datos desde formularios o sistemas externos.

7.4 uses = { Mapper.class }

Objetivo:
Permitir que un mapper use otros mappers auxiliares para delegar conversiones o submapeos.

Cómo funciona:

- Se declara con **@Mapper(uses = {OtroMapper.class})**
- MapStruct buscará métodos de conversión en esos mappers auxiliares.
- Ideal para modularizar y evitar duplicar lógica de mapeo.

Ejemplo: usar un **DateMapper** dentro de **OrderMapper**

DateMapper.java

```java
@Mapper(componentModel = "spring")
public interface DateMapper {
    @Named("asString")
    default String asString(LocalDate date) {
        return date != null ? date.toString() : null;
    }

    @Named("asDate")
    default LocalDate asDate(String date) {
        return date != null ? LocalDate.parse(date) : null;
    }
}
```

OrderMapper.java

```java
@Mapper(componentModel = "spring", uses = { DateMapper.class })
public interface OrderMapper {

    @Mapping(source = "orderDate", target = "orderDate", qualifiedByName = "asString")
    OrderDTO toDTO(Order order);

    @Mapping(source = "orderDate", target = "orderDate", qualifiedByName = "asDate")
    Order toEntity(OrderDTO dto);
}
```

Resultado:

- **OrderMapper** reutiliza los métodos de **DateMapper.**
- Permite un diseño modular y fácil de mantener.

7.5 @MappingTarget

Objetivo:
Modificar o actualizar un objeto destino existente en lugar de crear uno nuevo.
Ideal para operaciones tipo update, cuando quieres mapear un DTO sobre una entidad persistente existente.
Cómo funciona:

- Se usa como parámetro de un método de mapeo (**@MappingTarget**).
- MapStruct actualiza el objeto en lugar de instanciarlo.

Ejemplo: actualizar una entidad existente

```
@Mapper(componentModel = "spring")
public interface CustomerMapper {

    void updateEntityFromDTO(CustomerDTO dto, @MappingTarget Customer entity);
}
```

Uso en servicio:

```
public Customer update(Long id, CustomerDTO dto) {
    Customer entity = repository.findById(id).orElseThrow();
    customerMapper.updateEntityFromDTO(dto, entity);
    return repository.save(entity);
}
```

Resultado:

- Actualizas solo los campos del **DTO**, manteniendo el resto intactos (como id o relaciones JPA).
- Muy útil para endpoints **PUT** y **PATCH**.

7.6 defaultValue / ignore

Objetivo:
Controlar el mapeo de campos nulos o irrelevantes.

Cómo funciona:

- **defaultValue** asigna un valor si la fuente es null.
- **ignore = true** omite un campo en el mapeo (ni se copia ni se modifica).

Ejemplo:

```java
@Mapper(componentModel = "spring")
public interface OrderMapper {

    @Mapping(target = "id", ignore = true)
    @Mapping(target = "status", defaultValue = "PENDING")
    Order toEntity(OrderDTO dto);
}
```

Resultado:

- id nunca se mapea (lo gestiona JPA).
- Si status es null en el DTO, se asigna "**PENDING**" por defecto.
- Controlas exactamente qué campos se afectan durante la conversión.

8. Buenas prácticas MapStruct + Spring Boot

- ➢ Usa siempre componentModel = "spring".
- ➢ Define DTOs específicos por cada capa o contexto.
- ➢ Separa mapeos complejos en mappers auxiliares (uses).
- ➢ Evita lógica de negocio en mappers, usa @AfterMapping solo para cálculos simples.
- ➢ Versiona los DTOs si tu API evoluciona (v1, v2, etc.).
- ➢ Aprovecha List y Page mapeados automáticamente.